REMARKS

Claims 1-30 are pending in the present application. Reconsideration of the claims is respectfully requested.

I. Interview

Applicants thank Examiner Ali for the courtesy of an interview on February 23, 2004. Applicant argued that Lueh et al. does not teach determining if the portion of computer code is currently being compiled and redirecting the call to an interpreter, if the portion of computer code is currently being compiled. Examiner Ali agrees that Lueh does not teach this feature. Examiner Ali indicted that he will reconsider the rejection upon further search and consideration.

II. Drawings

The Office Action objects to the drawings for containing informalities. Applicants respectfully submit that the objection is based upon drawings that were submitted on September 28, 2000. Applicants further submit that formal drawings were submitted on November 22, 2000 and received in the U.S. Patent and Trademark Office on November 27, 2000. Accordingly, Applicants respectfully request withdrawal of the objection to the drawings.

III. 35 U.S.C. § 102, Alleged Anticipation, Claims 1-7, 11-17 and 21-27

The Office Action rejects claims 1-7, 11-17 and 21-27 under 35 U.S.C. § 102(e) as being allegedly anticipated by Lueh et al. (U.S. Patent No. 6,158,048). This rejection is respectfully traversed.

As to claims 1, 11-17 and 21-27, the Office Action states:

> As per claim 1, Lueh discloses a method of calling a portion of computer code in a multithreaded environment, comprising:
> receiving a call to the portion of computer code (col. 6 line 23 –

col. 7 linc 14, "Initially, at step 410, code selector takes a bytecode or expression from the code stream");

determining if the portion of computer code is currently being compiled (col. 6 line 23 – col. 7 linc 14, "Before selecting code for the bytecode, the selector looks ahead in the stream to see whether the expression starting at this bytecode matches one already associated with a scratch register"); and

redirecting the call to an interpreter, if the portion of computer code is currently being compiled (col. 6 line 23 – col. 7 line 14, "If the expression comparison at step 430 indicates that the expression are syntactically identical, then the code selector pushes the contents of the register onto the stack at step 450", wherein if the call matches code that is currently being compiled, the expression is redirected such that the compiled expression is used as the result of the subsequent code).

Office Action dated November 24, 2003, pages 2-3.

Claim 1, which is representative of the other rejected independent claims 11 and 21 with regard to similarly recited subject matter, reads as follows:

1. A method of calling a portion of computer code in <u>a multithreaded environment</u>, comprising:

receiving a call to the portion of computer code;

<u>determining if the portion of computer code is currently being compiled; and</u>

<u>redirecting the call to an interpreter, if the portion of computer code is currently being compiled</u>. (emphasis added)

A prior art reference anticipates the claimed invention under 35 U.S.C. § 102 only if every element of a claimed invention is identically shown in that single reference, arranged as they are in the claims. In re Bond, 910 F.2d 831, 832, 15 U.S.P.Q.2d 1566, 1567 (Fed. Cir. 1990). All limitations of the claimed invention must be considered when determining patentability. In re Lowry, 32 F.3d 1579, 1582, 32 U.S.P.Q.2d 1031, 1034 (Fed. Cir. 1994). Anticipation focuses on whether a claim reads on the product or process a prior art reference discloses, not on what the reference broadly teaches. Kalman v. Kimberly-Clark Corp., 713 F.2d 760, 218 U.S.P.Q. 781 (Fed. Cir. 1983). Applicants respectfully submit that Lueh does not identically show cach and every feature arranged as they are in the claims. Specifically, Lueh does not teach determining if the portion of computer code is currently being compiled and redirecting the call to an interpreter, if the portion of computer code is currently being compiled.

Lueh is directed to a method for eliminating common subexpressions from JAVA byte codes. In the system of Lueh, a code stream containing sequences of computer code is loaded into computer memory. The expression value for a first expression of the code sequence is computed and stored in a memory location. A tag is assigned to the memory location holding this expression value. This tag tracks which expression sequence value is held in each memory location. As code compilation continues, the code selector looks ahead in the code stream to see if any upcoming expression sequences already have an expression value stored in a memory location. The code selector compares the expression of a second code sequence with the code sequences annotated by the tags of expression values currently stored in memory. If the second code sequence matches a sequence already associated with a memory location, then the value of the matched sequence is pushed from the memory location onto a stack, and the computations of the expression of the second code sequence is skipped. If the second expression does not match any of the expressions represented by the tags, the expression value of the second expression is calculated and stored in a memory location. This memory location is then annotated with its own expression tag for future comparisons with upcoming expressions in the code stream.

Thus, with the system of Lueh, a determination is made as to whether an expression of a second code sequence is the same as an expression value already stored in memory. There is no teaching anywhere in the Lueh reference as to determining if the portion of computer code is currently being compiled. The Office Action alleges that this feature is taught by Lueh at column 6, line 23 to column 7, line 14, which reads as follows:

> Eliminating Common Subexpressions
>
> FIG. 4 is illustrates a flow diagram of the common subexpression elimination method of the present invention during the code generation process. Initially, at step 410, code selector takes a bytecode or expression from the code stream. Before selecting code for the bytecode, the selector looks ahead in the stream to see whether the expression starting at this bytecode matches one already associated with a scratch register. First, the compiler checks to see if any expressions are stored in the registers at step 420. If no stored expressions are available, the code selector selects an instruction sequence for the bytecode at step 460. Then the expression value is stored in a register at step 462 and the expression tag of the

> register containing the result of the instruction sequence is updated. The code selector reaches step 470 and checks to see if the end of the code stream has been reached. If it has not, then the code selector goes back to step 410 for the next bytecode. In the present embodiment, the code selector performs step 480 between the process of the code selector looping from step 470 to step 410. Step 480 checks to see if any of the stored values have changed in value. In other words, the check is to see if the availability of any stored expression are now invalid. Stored expression may become invalid when (1) instructions modify the value of the register R that contains the computed expression E, or (2) assignments or method calls modify a source value that is part of expression E. If the code selector discovers that a stored value has changed, then the affected stored value is killed at step 482. Killed in this sense means that the stored expression value is discarded. Then the code selector continues on back to step 410.
>
> But if at step 420, the code selector finds that stored expression values are available in registers, the selector then compares the current expression with a stored expression at step 430. The registers are checked in decreasing order of subsequence length to match the largest sized expression. If the comparison fails and the code selector finds that not all the stored expressions have been compared at step 440, then the selector loops back to step 430 and compares the current bytecode with the next largest expression in the registers. If there are no more stored expressions at step 440, the code selector selects an instruction sequence for the bytecode at step 442. As before the expression value is stored in a register at step 444 and the expression tag of the register containing the result of the instruction sequence is updated. Then the code selector reaches step 470 and checks to see if the end of the cod stream has been reached. If it has not, then the code selector loops back to the step 410 for the next bytecode.
>
> If the expression comparison at step 430 indicates that the expression are syntactically identical, then the code selector pushes the contents of the register onto the stack at step 450. Then at step 452, the code selector skips over the common subexpression in the bytecode stream. Now the selector reaches step 470 as in the other paths described earlier. If the end of the code stream has not been reached, the code selector returns to step 410 for the next bytecode in the code stream. If the stream has ended, the phase is completed.

In this section, Lueh is describing a method where a compiler checks to see if any expressions are stored in the registers. If not, the code selector takes an expression from the code stream, stores the expression in a register and updates the expression tag of the register containing the result of the instruction sequence. Then the selector looks ahead in the stream for the next expression and compares the current expression with stored

expressions. If the expression comparison indicates that the expressions are syntactically identical, then the code selector pushes the contents of the register onto a stack and skips the current expression. If the comparison fails, the code selector selects an instruction sequence for the expression, stores the expression value in a register and updates the expression tag of the register containing the result of the instruction sequence. Thus, the system of Lueh is concerned with eliminating common expressions by using stored instruction sequences for common expressions, thereby optimizing JAVA code generation. Lueh is not concerned with determining if the portion of computer code is <u>currently</u> being compiled in a multithreaded environment. That is, Lueh compares incoming code to stored values in <u>memory</u>, not what is currently being compiled in the processor.

Lueh also does not redirect the call to an interpreter, if the portion of computer code is currently being compiled. Lueh merely teaches that in the event that the expression comparison indicates that the expression are syntactically identical, then the code selector pushes the contents of the register onto the stack and the current expression being compared to the register value <u>is skipped</u>. Even if Lueh were able to determine if the portion of computer code was currently being compiled, which it does not, the teachings of Lueh would discard the incoming call and send a register value to a stack. Additionally, a stack is not an interpreter, as the Office Action appears to indicate. Thus, Lueh does not teach redirecting the call to an interpreter, if the portion of computer code is currently being compiled.

Furthermore, Lueh does not mention a multithreaded environment as in the presently claimed invention. Without a multithreaded environment, there can be <u>no</u> determination of whether the portion of the code is <u>currently</u> be compiled, because there is no other thread in which the compiler could be running. Lueh clearly teaches, "the method of the present invention first loads <u>a code stream</u> containing sequences of computer code into computer memory" (see column 2, lines 18-20). There is no teaching in Lueh to compare the incoming calls from multiple code streams. Therefore, in addition to Lueh not teaching determining if the portion of computer code is <u>currently</u> being compiled and redirecting the call to an interpreter if the portion of computer code is currently being compiled, Lueh is not directed to the same field of invention as the

presently claimed invention as Lueh does not teach implementation in a multithreaded environment.

Thus, Lueh does not teach each and every feature of independent claims 1, 11 and 21 as is required under 35 U.S.C. § 102(e). At least by virtue of their dependency on independent claims 1, 11 and 21, respectively, Lueh does not teach each and every feature of dependent claims 2-7, 12-17 and 22-27. Accordingly, Applicants respectfully request withdrawal of the rejection of claims 1-7, 11-17 and 21-27 under 35 U.S.C. § 102(e).

Furthermore, Lueh does not teach, suggest, or give any incentive to make the needed changes to reach the presently claimed invention. Absent the Examiner pointing out some teaching or incentive to implement Lueh to determine whether the portion of computer code is currently being compiled and redirect the call to an interpreter if the portion of computer code is currently being compiled in a multithreaded environment, one of ordinary skill in the art would not be led to modify Lueh to reach the present invention when the reference is examined as a whole. Absent some teaching, suggestion, or incentive to modify Lueh in this manner, the presently claimed invention can be reached only through an improper use of hindsight using the Applicants' disclosure as a template to make the necessary changes to reach the claimed invention.

Moreover, in addition to their dependency from independent claims 1, 11 and 21, respectively, Lueh does not teach the specific features recited in dependent claims 2-7, 11-17 and 21-27. For example, with regard to claims 3, 13 and 23, Lueh does not teach redirecting the call to an interpreter includes redirecting the call to a Java Virtual Machine Interpreter such that the portion of computer code is interpreted by the Java Virtual Machine Interpreter in response to receiving the call to the portion of computer code. The Office Action alleges that this feature is taught at column 5, line 43 to column 6, line 22, which reads as follows:

> An example of running a Java program in a networked computer environment is provided with reference to FIG. 2 and FIG. 3. FIG. 2 illustrated the steps of downloading, compiling, and running a Java program in a Java Virtual Machine that compiles the code before execution. FIG. 3 illustrates a block diagram of the elements in a client computer system 300 equipped with a Java Virtual Machine to interpret and compile Java class files. The client computer system 300 includes

computer hardware 310 controlled by an operating system 320. The computer hardware further comprises of computer memory 312 and machine registers 314. The system 300 also includes a Java Virtual implementation 330 for running Java class files 360. In the present embodiment, the Java Virtual Machine 330 running on the client computer system 300 relies on services from the underlying operating system 320 and the computer hardware 310. Furthermore, the Java Virtual Machine 330 may utilize a Java Interpreter 332 for interpreting Java class files 360 or a Java "Just-In-Time" (JIT) computer 334 to generate compiled Java code.

In a networked environment, a user would first access a computer server through the network and download the desired Java class file 360 into a client computer system 300 as in step 210. After the class file 360 has been downloaded, the Java Virtual Machine 330 verifies the class file at step 220. Step 220 of verifying the class file is to ensure that the program will not cause security violations not will it cause harm to the computer system resources. After the Java program has been verified, a Java JIT compiler 334 compiles the Java class file and generates compiled Java code 340 in the form of native processor code at step 230. Then this compiled Java code 340 is directly executed on the computer hardware 310 at step 340. <u>Java programs which have been compiled and translated into the native code of the processor in the client system 300 execute much faster than Java programs that executed using a Java interpreter</u>. In order to maintain the state of the Java Virtual Machine 300 and make system calls, the compiled Java code 340 makes calls 350 into the Java Virtual Machine 330.

Although the above example describes the distribution of a Java class file via a network, Java programs may be distributed by way of other computer readable mediums. For instance, a computer program may be distributed through a computer readable medium such as a floppy disk, a CD ROM, a carrier wave, or even a transmission over the internet. (empahsis added)

Though this section of Lueh may teach that a Java Virtual Machine may use a Java interpreter, Lueh teaches away from using a Java Interpreter by implementing a method where computer code is compiled, stored in registers compared to incoming expressions and if the comparison is syntactically identical, using the stored register value. In fact, Lueh clearly teaches "<u>Java programs which have been compiled and translated into the native code of the processor in the client system execute much faster than Java programs that executed using a Java interpreter</u>". Thus, Lueh does not teach or fairly suggest the specific features as recited in claims 3, 13 and 23.

As an additional example, with regard to claims 5, 15 and 25, Lueh does not teach redirecting the call is performed in response to a Just-In-Time (JIT) invoker field, in a control block of the portion of computer code, pointing to a JIT to Java Virtual Machine (JVM) routine. The Office Action alleges that this feature is taught by Lueh at column 5, line 43 to column 6, line 22, shown above. Nowhere in this section, or any other section of Lueh, is a JIT Invoker even mentioned. As discussed above, Lueh is directed to an enhanced JIT Compiler that, ideally, would avoid the need for a JIT Interpreter. Conventionally, the purpose of a JIT Invoker is to control calls to be sent to a JIT Compiler. Thus, not only does Lueh not teach a JIT Invoker, there is no suggestion of using a JIT Invoker to redirect calls to an interpreter in Lueh. Therefore, Lueh does not teach the specific features as recited in claims 5, 15 and 25.

As a further example, with regard to claims 6, 16 and 26, Lueh does not teach determining if compilation of the portion of computer code has ended and redirecting the call to a compiled version of the portion of computer code if the compilation of the portion of computer code has ended. The Office Action alleges that this feature is taught by Lueh at column 6, line 23 to column 7, line 14, shown above. As shown above, Lueh teaches comparing an incoming expression to a value of an expression that is <u>stored in a register</u>. Nowhere in this section, or any other section of Lueh, is it taught to determine if compilation of the portion of computer code has ended and redirect the call to a compiled version of the portion of computer code if the compilation of the portion of computer code has ended. Thus, Lueh does not teach the specific features recited in claims 6, 16 and 26.

Therefore, in addition to being dependent on independent claims 1, 11 and 21, respectively, dependent claims 2-7, 12-17 and 22-27 are also distinguishable over Lueh by virtue of the specific feature recited in these claims. Accordingly, Applicants respectfully request withdrawal of the rejection of claims 2-7, 12-17 and 22-27 under 35 U.S.C. § 102(e).

IV. 35 U.S.C. § 103, Alleged Obviousness, Claims 8, 18 and 28

The Office Action rejects claims 8, 18 and 28 under 35 U.S.C. § 103(a) as being allegedly unpatentable over Lueh et al. (U.S. Patent No. 6,158,048) in view of Levine (U.S. Patent No. 6,311,325 B1). This rejection is respectfully traversed.

Applicant submits that Levine fails to teach or suggest the features alleged in the Office Action. In addition, the Levine patent and the instant application were, at the time of the invention was made, owned by, or subject to an obligation of assignment to the same person. 35 U.S.C. § 103(c) states:

> (c) Subject matter developed by another person, which qualifies as prior art only under one or more of subsections (e), (f), and (g) of section 102 of this title, shall not preclude patentability under this section where the subject matter and the claimed invention were, at the time the invention was made, owned by the same person or subject to an obligation of assignment to the same person.

The instant application was filed on or after November 29, 1999. The Levine patent qualifies as prior art only under 35 U.S.C. § 102(e). And, the instant application and the Levine patent were commonly owned or subject to an obligation of assignment to the same person at the time the invention was made. Therefore, the Levine patent cannot be used in a 35 U.S.C. § 103 rejection to preclude patentability. As such, the rejection is improper and should be withdrawn. Claims 8, 18 and 28 recite subject matter addressed above with respect to claims 1, 11 and 21 and are allowable for the same reasons.

In view of the above, Applicants respectfully submit that neither Lueh nor Levine, either alone or in combination, teach or suggest the features of claims 8, 18 and 28. Accordingly, Applicants respectfully request withdrawal of the rejection of claims 8, 18 and 28 under 35 U.S.C. § 103(a).

V. 35 U.S.C. § 103, Alleged Obviousness, Claims 9-10, 19-20 and 29-30

The Office Action rejects claims 9-10, 19-20 and 29-30 under 35 U.S.C. § 103(a) as being allegedly unpatentable over Lueh et al. (U.S. Patent No. 6,158,048) in view of Levine (U.S. Patent No. 6,311,325 B1) and further in view of Beadle et al. (U.S. Patent No. 6,324,687 B1). This rejection is respectfully traversed.

Applicant submits that Beadle fails to teach or suggest the features alleged in the Office Action. In addition, the Beadle patent and the instant application were, at the time of the invention was made, owned by, or subject to an obligation of assignment to the same person. 35 U.S.C. § 103(c) states:

> (c) Subject matter developed by another person, which qualifies as prior art only under one or more of subsections (e), (f), and (g) of section 102 of this title, shall not preclude patentability under this section where the subject matter and the claimed invention were, at the time the invention was made, owned by the same person or subject to an obligation of assignment to the same person.

The instant application was filed on or after November 29, 1999. The Beadle patent qualifies as prior art only under 35 U.S.C. § 102(e). And, the instant application and the Beadle patent were commonly owned or subject to an obligation of assignment to the same person at the time the invention was made. Therefore, the Beadle patent cannot be used in a 35 U.S.C. § 103 rejection to preclude patentability. As such, the rejection is improper and should be withdrawn. Claims 9, 10, 19, 20, 29 and 30 recite subject matter addressed above with respect to claims 1, 11 and 21 and are allowable for the same reasons.

In view of the above, Applicants respectfully submit that neither Lueh, Levine nor Beadle, either alone or in combination, teach or suggest the features of claims 9, 10, 19, 20, 29 and 30. Accordingly, Applicants respectfully request withdrawal of the rejection of claims 9, 10, 19, 20, 29 and 30under 35 U.S.C. § 103(a).

VI. Conclusion

It is respectfully urged that the subject application is patentable over the prior art of record and is now in condition for allowance. The Examiner is invited to call the undersigned at the below-listed telephone number if in the opinion of the Examiner such a telephone conference would expedite or aid the prosecution and examination of this application.

Respectfully submitted,

DATE: February 23, 2004

Francis Lammes

Francis Lammes
Reg. No. P-55,353
Carstens, Yee & Cahoon, LLP
P.O. Box 802334
Dallas, TX 75380
(972) 367-2001
Agent for Applicants

SRT/fl